Optimisation of business process tenant distribution in the Cloud with a genetic algorithm

Guillaume Rosinosky[1,2], Samir Youcef[2], and François Charoy[2]

[1] Bonitasoft, Grenoble, France,
guillaume.rosinosky@bonitasoft.com,
http://www.bonitasoft.com
[2] Inria Nancy Grand Est - Université de Lorraine - CNRS

Abstract. With the generalization of the Cloud, software providers can distribute their software as a service without investing in large infrastructure. However, without an effective resource allocation method, their operation cost can grow quickly, hindering the profitability of the service. This is the case for BPM as a Service providers that want to handle hundreds of customers with a given quality of service. Since there are variations in the capacity and the number of users, the allocation method must be able to adjust the resource and the allocation of customer on these resources. In this paper we present a cost optimization model and a heuristic based on genetic algorithms to adjust resource allocation to the need of a set of customers with varying BPM task throughput. Experimentations using realistic customer loads and cloud resources capacities shows the gain of this method compared to previous approaches.

Keywords: elasticity, BPM, cloud, genetic algorithm

1 Introduction

From a customer point of view, consuming "Business Process Management as a Service" (BPMaaS) delivered by a service provider has advantages that IT people widely acknowledge. It reduces the operational burden and allows to rely on the provider for all the maintenance and provisioning of the service and to concentrate on the business dimension. From the provider point of view, it increases the operational complexity. The provider must ensure that all his customers receive the same attention and the same quality of service at all time. He must also ensure that he operates at the best possible cost. Public cloud providers bill their resources on demand. The service provider wants to pay only for what is really needed for his customers. The Business process service provider must provide the best service possible, according to the customer service level agreement (SLA) while reducing the cost from the public cloud use. A Business Process Management System (BPMS) deployment is complex. It includes application servers, process engines and database management systems. If the BPMaaS is

multi-tenant, it can support several customers on the same software installation. Clustered installation can be deployed for high availability. Usage pattern for customers are very diverse and the BPM task throughput evolves among the day and the week. It is possible to distribute customers processes on different deployments on different cloud compute instances in order to maintain the cost of the overall infrastructure. In order to optimize the placement of customers over the day, we can migrate them from an installation to another. For instance, moving a customer from one expensive installation but powerful set of instances to a cheaper one, but able to support less operations can be financially interesting. However, migrations are critical since they can generate short disruption of service on the customer side. This is mostly due to the migration of the process data and of the executing processes We must control them and limit their number.

In our previous work [15], we proposed an original strategy that relies on time series segmentation on one side, and on iterative usage of our time slot heuristic [14] on the other side, based on load evolution to select the best migration time. The resulting algorithm has two levels : one for finding the best migration strategy (a matrix representing for each tenant the precise time slots where they should be migrated), and an iterative algorithm with this information. The strategy provided better results than the previous one and was more effective in term of performances. We could deal with a high number of tenants without performance issues. However results could be enhanced compared to those obtained with an integer linear model solver. Here we propose a new integer linear programming (ILP) model as an alternative of the iterative heuristic, and a genetic algorithm that aims at finding the cheapest migration strategy. We obtain better results, and, as we will see in the experiment part, an insight based on splitting groups of tenants for obtaining even cheaper configurations.

The next section of this paper is a state of art related to BPM elasticity in the cloud. In the third section we describe our migration strategy model, our specific genetic algorithm approach coupled to a solving of the model, and our iterative heuristic. We study experimental results on both of the approaches compared to the previous results and show how we achieve a surprising experimental result. The last part concludes and presents the future work.

2 Related work

A lot of work has been devoted to elasticity in the cloud and more precisely elasticity for BPM or orchestration systems. Schulte and al. [16] did a general review on the topic and gives direction for future research. Here, we focus on the resource allocation and scheduling parts and use a tenant-centric approach based on BPM task throughput, instead of the BPM process-centric from other approaches. We considers horizontal elasticity (adding or removing resources) and vertical elasticity (using lower or higher end resources) at the same time.

Though not cloud-related, Djedovic et al. proposes in [4] a genetic algorithm for BPM task scheduling to their corresponding resources. It is based on a repre-

sentation of each corresponding resource. Their goal is to minimize the waiting time and the global resource cost. Rekik et al. [13] propose an integer programming model based on general hardware metrics for BPM elasticity on the cloud. Their approach is based on resource allocation and BPM task scheduling, though very interesting, does not tackle multiple time slots, data migration or multi-tenancy. Other attempts on BPM elasticity such as [9], [10], [5], [18] do not tackle multi-tenancy or migration cost.

Numerous attempts on virtual machines has been done such as those of the machine reassignment problem [1] who treats the migration cost. [7], [2] answered to this problematic, but these approaches use hardware metrics, and aggregate migration cost in the objective function. Moreover, virtual machine allocation is different of our problem as the hardware is already defined.

This paper is an evolution of our previous work [15]. We described our approach based on time series segmentation for deducing the good time slot to migrate tenants, and on the iterative use of an enhanced version of our time slot heuristic. We also presented the corresponding ILP (Integer Linear Programming) model. Results were encouraging compared to an intuitive approach, but could be improved regarding the results that we obtain with a solver. In the next section we present our new approach that provides better results and enhance scalability.

3 Approach and model

In this section we present our fixed migration strategy model and our optimization method that is based on a genetic algorithm.

Our approach is tenant-centric i.e we distribute customers (or tenants) as a whole on BPM installations and their cloud resources. We consider that it is easier and more realistic to manage deployment by customer rather than by process since we consider the process, the business data and security configuration that is specific for a tenant and that may be shared between executions. Our assumptions are the following :

- a tenant is a customer of the BPMaaS. Tenants run BPM processes composed of BPM tasks. In order to execute them, the BPMS needs computing power, network bandwidth, disk and memory. It also relies on relational databases for the persistence part and load balancers for clustered installations. These metrics are difficult to relate to the business activity. Thus we adopt the BPM task throughput as our main performance metric. It corresponds to the number of completed BPM tasks for one period of time. As BPM task states are persisted with transactions, this metric can be used for the whole system. It is also meaningful for the customers. We assume that we know in advance the BPM task throughput per second for each tenant and each time slot. Our approach is offline.
- a cloud resource is one or several Cloud compute instances that we can use for the database tier, BPM system tier, load balancer tier, etc. It is able to support a full BPMS installation. Multiple cloud resource types correspond

to several types of IaaS cloud compute instance. An example on Amazon Web Service[3] (AWS) would be a combination of a *r3.large* for the persistence tier, and *c4.large* for the BPMS tier. We assume that there is a price per time slot for each cloud resource type. A cloud resource type, and by extension a cloud resource has a capacity expressed in BPM task throughput.

- a cloud resource is able to host multiple tenants. For each time slot, the sum of the task throughputs of a cloud resource's tenants must not exceed the capacity of the cloud resource. Each tenant must be assigned to an active cloud resource at each time slot.
- we name migrations the action of moving the tenant data and process from one cloud resource to another. It generates QoS breaks for the customers [3]. Thus, we limit the number of migrations for each tenant to a defined value.

We proposed in [15] a method based on an iterative heuristic that can gives the list of cloud resources required and a mapping of tenants on each resource, this for each time slot. We present it in the next section.

3.1 First allocation heuristic

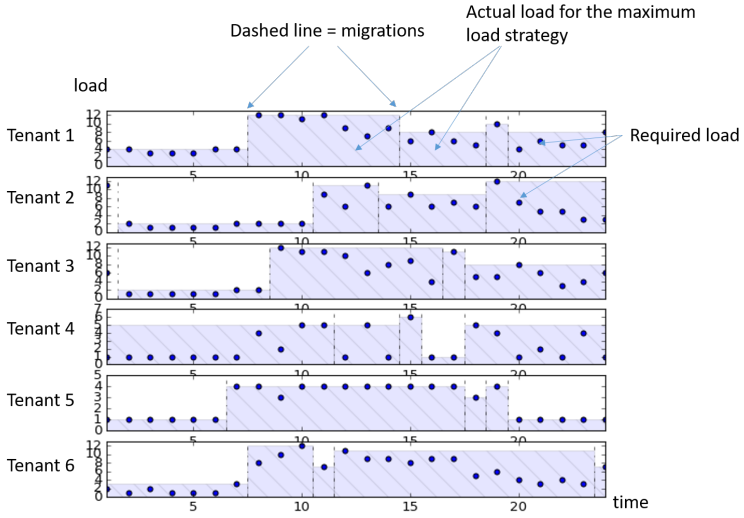

Fig. 1. Exemple of a migration strategy

Our first heuristic has two parts : first we have to choose a *migration strategy* i.e the time slots where each tenant can migrate, and second apply an heuristic

[3] https://aws.amazon.com/

using this migration strategy in order to obtain, the cloud resources and the placement of tenants on it. An example of migration strategy is shown in figure 1. In order to select a cheap migration strategy, we used time series segmentation [11]. It allowed to select good migration times for each tenant. It provides better results than an intuitive approach, but the comparison with the optimal solution computed with a solver shows that it is far from the optimal cost. The new approach we propose in this paper gives better results. We propose to use a genetic algorithm able to find migration strategies that reduces the resource cost, and an alternative to the iterative timeslot heuristic based on integer linear programming. We present the latter in the following section.

3.2 A efficient model for migration strategies

Let the following variables :

- $\mathcal{T}$, the set of cloud configuration types, with t its cardinality.
- $\mathcal{I}$, the set of tenants with n its cardinality
- $\mathcal{J}$, is $\mathcal{T} \times \mathcal{I}$ the set of all possible cloud configurations associated with each tenant. its cardinality is $m = t \times n$
- C_j, and W_j, respectively the cost and the capacity for the configuration j, with j in $\mathcal{J}$
- w_i(k), the required capacity for the tenant i during time slot k
- $\mathcal{K}$ defines all the time slots, from 0 to D, where $D+1$ is the number of time slots.
- ${x_j}^i(k)$, the assignment of tenant i to configuration instance j during time slot k
- $y_j(k)$, the activation of configuration j during time slot k
- M, the maximum number of migrations of tenants between cloud resources on all time slots
- $h_i(k)$ with $0 \leq k \leq D-1$. $h_i(k)$ is equal to 0 if the tenant i is not allowed to be migrated between time slot k and $k+1$, and equal to 1, if it is allowed. The set of all $h_i(k)$ (for each tenant and each time slot) is a migration strategy.
- migration strategies assume the maximum number of migrations allowed per tenant : $\forall i \in \mathcal{I} \sum_k^{k \in \mathcal{K}} h_i(k) = M$ where M is the number of migrations.

The objective for our model is to minimize the total cost for all active cloud resources, for each time slot.

$$\min \sum_j^{j \in \mathcal{J}} \sum_k^{k \in \mathcal{K}} C_j y_j(k) \tag{1}$$

We must ensure that the following constraints are not violated

$$\forall i \in \mathcal{I}, \forall k \in \mathcal{K} \sum_j^{j \in \mathcal{J}} {x_j}^i(k) = 1 \tag{2}$$

$$\forall j \in \mathcal{J}, \forall k \in \mathcal{K} \sum_{i}^{i \in \mathcal{I}} w_i(k) {x_j}^i(k) \leq W_j y_j(k) \tag{3}$$

$$\forall j \in \mathcal{J}, \forall i \in \mathcal{I}, \forall k \in \mathcal{K} | h_i(k) = 0, {x_j}^i(k) = {x_j}^i(k+1) \tag{4}$$

$$\forall i \in \mathcal{I}, \forall j \in \mathcal{J}, \forall k \in \mathcal{K}, {x_i}^j(k) \in \{0,1\}, y_j(k) \in \{0,1\} \tag{5}$$

Equation 2 represents the obligation of a tenant to be placed at each time slot on an active cloud resource. Equation 3 means that the sum of the required capacity for each tenant on one cloud resource cannot exceed the capacity of the cloud resource. Equation 5 represents the variables we use, one representing the activity of a cloud resource at time (y), and the other representing the mapping of each tenant on each cloud resource at each time (x).

Equation 4 represent the migration strategy. The equality constraint means that for a tenant i and a time slot k, assignation values ${x_i}^j(k)$ will stay the same on time slots k and $k+1$. When a tenant is authorized to migrate between resource, there is no constraint for this tenant. Generalizing this on all resources produces the desired effect, and enforces the maximum number of migrations constraint we need.

As the iterative heuristic, this model needs a pre-defined migration strategy. We present in the next sub section our genetic algorithm approach, who aims to find the cheapest one.

3.3 Cost optimization via genetic algorithms

A genetic algorithm is a meta-heuristic belonging to the family of evolutionary algorithms, and inspired by natural selection [17]. Its principle is to produce directed random evolutions on a population of individuals until it obtains one or several individuals with a good fitness value. Here is a description of the different steps :

1. generation of an initial population of different individuals (also named *chromosomes*).
2. evaluation of the fitness of the different individuals (in our case the cost)
3. selection of the individuals to use for computing future generations (parents)
4. offspring generation using the selected parents, mixing elements from each one in the resulting individuals
5. random mutations application on the offspring
6. fitness evaluation of the resulting offspring
7. survivors selection between the offspring and the original population
8. process stop after a number of generations or a time limit or go back to step 3.

For each step, there are multiple approaches. In the following, we describe the solution we have designed.

Individual representation We want to find the best migration strategy for all the tenants and time slots. To represent an individual, we vectorize a migration strategy by concatenating migration strategies of each tenant (each one corresponding to a vector of D boolean values). The size of the vector will be $|D| \times |I|$, with each element being equal to zero or one. For instance with two tenants and three time slots, the first migrating on the second time slot and the second tenant on the third time slot, we will have the following representation : $[\mathbf{0, 1, 0, 0, 0, 1}]$.

Population initialization We initialize the population with all the segmentation algorithm combinations, and with random individuals with the correct number of mutations for each tenant.

Fitness evaluation We want to find the migration strategies that produces the cheapest cost. The fitness score corresponds to the total cost of all the active resources on the time slots. To evaluate it on the different individuals, we compute the allocation and placement of the tenants on the cloud resources. In our case, we run our iterative time slot heuristic [15] or a solver on our model presented in section 3.2, for each individual. We keep the cloud resources and tenants assignation distribution in memory for the next steps, and of course the fitness score.

Parent selection For this step, we use a classical rank selection strategy. We sort the population by fitness and we select randomly, and with a higher priority, the individuals with the higher rank for parents.

A specific mutation : co-hosted tenant migration mutation strategy In classic approaches, mutation updates randomly individuals, depending on a mutation rate, switching scalar values from zero to one or the other way around [17]. In our case, we cannot use this approach, as the number of migrations for each tenant is bounded. We developed two alternative mutations more suited to our problem.

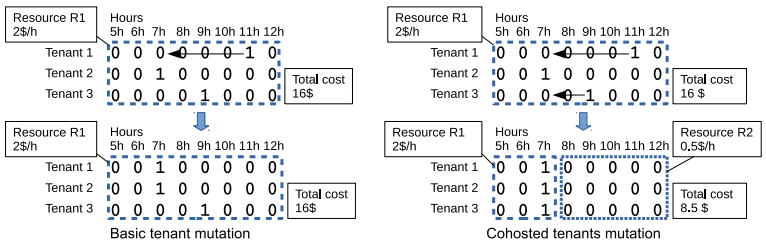

Fig. 2. Basic tenant mutation vs cohosted mutation

In the first one, we reassign to another hour some existing mutations. We consider a random tenant, a random origin migration authorization (value equal to one in the migration strategy matrix) and a random migration non authorized for the same tenant (value equal to zero in the migration strategy matrix). We authorized this on multiple tenants (i.e mutation points). However, this approach was not very effective in our case, and the gain was poor even after lots of iterations.

Our plan was to accelerate the convergence speed. The time slot algorithm [14], [15] uses a resource-based approach. Moving tenants located on the same resource at different time slots never frees the resource. In figure 2, we compare a basic tenant mutation and our co-hosted tenant mutation approach. We consider three tenants from 1 to 3, and hours as time slot, from 5am to 12pm. The tenants are initially placed on the resource R1, that costs 2$ per hour. For all the time slots it is possible to move all these tenants to the cheaper resource R2 - in this case, the sum of their current load permits it. Moving the authorization of migration of tenant 1 to 7 am will not be sufficient to free the expensive resource R1. Indeed, moving tenant 1 and tenant 2 to another cheaper resource would be more costly than to let all the tenants on the resource R1, as there would be two active resources. Even if this approach may be better in the long run, our iterative time slot heuristic considers time slots separately.

Our alternative approach consists in shifting the authorization to migrate for each co-hosted tenant at the designed time slot for the reference tenant's resource. For this, for each tenant, we iteratively move back the time slots until we find an authorization to migrate (or until we attain the beginning of the time slot space), and if we find one, we set it to zero while setting to one the "destination" time slot. If the "destination" time slot is already set to one, we ignore this behavior. The example on the right side of figure 2 describes this principle. There, it is possible to migrate all the tenants of resource R1 to the cheaper resource R2, and thus save money. We present this new behaviour in algorithm 1.

Offspring generation : the tenant crossover strategy The crossover technique we use consist in switching for some random tenants their migration time. First, two children identical to two migration strategies parents are generated. Then, depending on the *ncp* number of cut points, *ncp* tenants will see their migration hours switched in the children, as in figure 3.

Generational replacement We use a traditional approach where the entire population is replaced by the offspring, except for the best individuals from the original population (named elites). They replace the less fit offspring in the future population.

Termination condition We use a time limit end condition. This will allow us to compare different solutions based on this limit.

Algorithm 1 Cohosted mutation

```
1: procedure COHOSTED MUTATION(candidate, mutationRate, distributions, mutationPointsNumber, timeslotQuantity, tenantQuantity)
2:     if random(1) ≤ mutationRate then
3:         tenantsToMove ← ∅
4:         for mp in mutatationPointsNumber do
5:             concernedTimeSlot ← int(random(timeslotQuantity))
6:             destinationTimeslot ← int(random(timeslotQuantity))
7:             tenantToMove ← int(random(tenantQuantity))
8:             concernedResource ← distributions.getResource(concernedTenant, concernedTimeslot)
9:             cohostedTenants ← distributions.getTenants(concernedResource, concernedTimeslot)
10:            for tenantincohostedTenants do
11:                if candidates[tenant][destinationTimeslot] = 0 then
12:                    shiftTimeslot ← concernedTimeslot
13:                    while shiftTimeslot ≥ 0 do
14:                        shiftTimeslot ← timeslot − 1
15:                        if candidates[tenant][shiftTimeslot] = 1 then
16:                            candidates[tenant][shiftTimeslot] = 0
17:                            candidates[tenant][destinationTimeslot] = 1
18:                            break
    return candidate
```

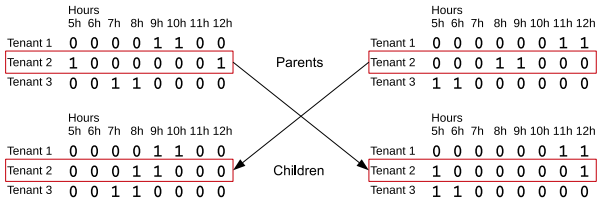

Fig. 3. Crossover heuristic

Adapted algorithm workflow We switched the mutation phase and the crossover phase. The cohosted mutation requires to have the cost of the migration strategy in the population. We compute the cost in the fitness evaluation phase. The crossover phase generates potentially unknown (not yet computed) migration strategies. Thus, we do it after the mutation phase. In our case, "parents" are mutated instead of the offspring.

We can use our technique both with our iterative heuristic [15] and with an optimization of our model presented in part 3.2 with the solver. In the next section, we present our experiments and the corresponding results.

4 Experimentation

Our experiment allows us to show that we enhance the results from our previous approach, and obtain good results with both our iterative heuristic [15] and

the model presented in section 3.2 when using our genetic algorithm approach presented in section 3.3.

We conducted tests with the same cloud resources price and size, and the same seeds than in our previous work [15]. We limit ourselves to configuration of two compute resources (see table 1).

DB inst. type	**AS inst. type**	**price**	**task TP**	**task TP per $**
db.m3.medium	m3.medium	0.177	16.400	92.656
db.m3.medium	c4.large	0.223	23.157	103.845
db.r3.large	c4.large	0.399	55.164	138.255
db.r3.large	c4.xlarge	0.518	58.067	112.100
db.r3.xlarge	c4.large	0.674	65.113	96.607
db.r3.large	c4.2xlarge	0.757	61.474	81.208
db.r3.xlarge	c4.xlarge	0.793	83.236	104.963
db.r3.xlarge	c4.2xlarge	1.032	89.149	86.384
db.r3.2xlarge	c4.2xlarge	1.587	105.794	66.663
db.r3.2xlarge	c4.4xlarge	2.063	107.585	52.150
db.r3.4xlarge	c4.4xlarge	3.173	115.283	36.332
db.r3.4xlarge	c4.8xlarge	4.126	129.279	31.332

Table 1. cloud configuration price, mean task throughput, and mean task throughput by dollar

For the customer part, we vary the number of tenants (10, 25, 50 and 100), and we use different throughputs of BPM task per second based on data from the BPMS BonitaBPM[4] customers. We use minimum and maximum throughput per second found in the anonymized execution history table (table 2). We have then generated each tenant's initial time slot load randomly following an uniform distribution between the two throughputs. Our next step was to generate the variation of load between time slots by adding or removing a random value limited to one quarter of the difference between the maximum and the minimum load (tenant gap of 0.25) as in [15].

customer	*days*	*minimum*	*maximum*
A	4	1	120
B	1	14	16
C	45	0	120
D	7	1	3
E	45	5	120
F	550	0	4

Table 2. For each customer, the day interval, the minimum and the maximum task throughput per second for each hour.

For our experiments, we used the python library Inspyred [6] for the genetic algorithm that integrate well with our environment.

[4] http://www.bonitasoft.com/

4.1 Experiment Parameters

In order to obtain significant and realistic results, we used the following parameters:

- four values for the number of tenants : 10, 25, 50 and 100
- time slot size of one hour as the cost model of compute instances of AWS
- we limit the allowed number of migration to 4 per day
- we consider a 2 days period (thus limiting migrations to 8)
- we limit the number of elites individuals to 5
- we chose a mutation rate of 0.4
- we chose a population size of 20
- we chose a number of mutation points correspond to the number of tenants divide by 5
- we limit the Genetic algorithm computation time to 600 or 1800 seconds
- we limit the solver computation time to 5 seconds

4.2 First experiment results

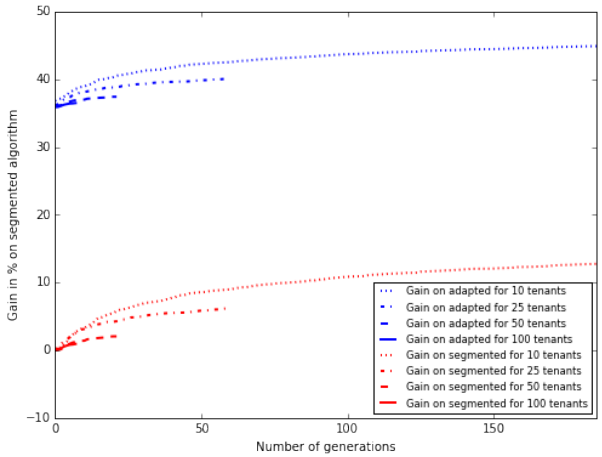

Fig. 4. Mean genetic algorithm gain on best initial segmented population for 600 second of running time

On figure 4 we show the relative gain of this new approach compared to our previous approach (segmentation) in red (in the upper part of the figure), and on the adapted strategy in blue (in the lower part of the figure). The *adapted strategy* corresponds to the intuitive approach where for each tenant, we book the cheapest resource able to withstand all the load of the studied time slots. This approach do not need migrations but can become very expensive. As expected,

the gain is better for 10 tenants than for 100 tenants since the system has more time to search for the cheapest solution. For 10 tenants, we obtain more than 10 % enhancement on the original approach, and more than 45 % on the intuitive approach. However for 100 tenants, we have only a 1 % enhancement.

It appears that either the iterative usage of the heuristic, the genetic algorithm or the two of them is more efficient for a small number of tenants for the same number of generations. This is why we conducted experiments where we apply the proposition to subsets of the tenants and we aggregated the results as described in the next section.

4.3 Splitting the tenants in subgroups or the splitting strategy

For this solution, we split the set of tenants in small groups selected randomly. We tested different size of splitted groups with various number of tenants and we applied the previous method keeping the same total computation time. Figure 5 shows the results we obtained with the genetic algorithm and the iterative heuristic. The x axis corresponds to the size of the groups of tenants. The y axis shows the relative gain compared to the results with no partition. A subset size with the same size as the number of tenants corresponds to no split, the gain is zero.

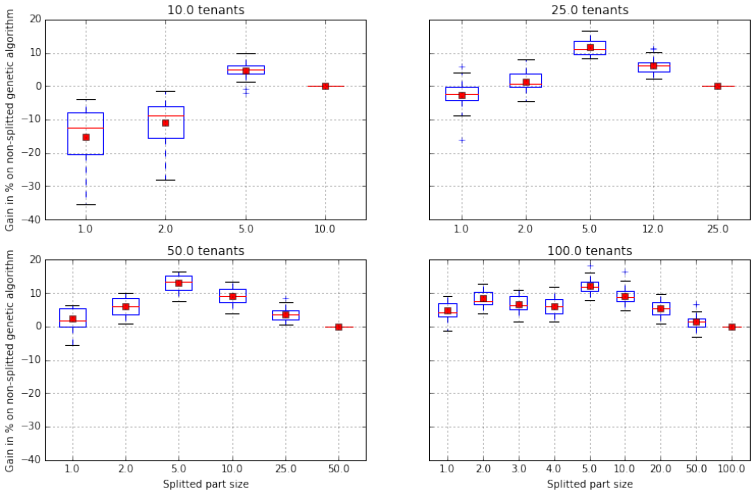

Fig. 5. Gain depending on splitting strategy for various split quantities.

We obtain the best results with partitions of 5 tenants in all cases. For the experiments we ran, the gain varies from 5% to 15%. We have no good explanation today for this result but it is reproducible. Our tests with the solver give the same results for the size of the groups than with the heuristic. In the next subsection we present our results with groups of 5 tenants.

4.4 Results for solver and iterative heuristic

We implemented our model (presented in subsection 3.2) using PuLP [12] with the Gurobi solver [8]. With only some parameters it is possible to launch the genetic algorithm against the heuristic or against the solver. However, for execution time and cost reasons, we were not able to test every set of parameters. For instance, with our current implementation, we managed to obtain results with the solver only up to a size of 25 tenants for the partition. Indeed, the duration of the initialization part and the required memory makes it impossible to run with more tenants. Thus we have limited our extensive tests to parts of 5 tenants, for a total of 50 and 100 tenants. However, as we can see, the results are interesting, and stay close to the results of the heuristic. Figure 6 shows the absolute gain we obtained, and the corresponding percentage compared to the *adapted approach* cost, for 600 seconds and 1800 seconds of running time. We also present the non-splitted result for the segmented approach (results of the previous paper), and the splitted segmented approach where we apply time series segmentation on the groups of 5 tenants instead of all the tenants simultaneously.

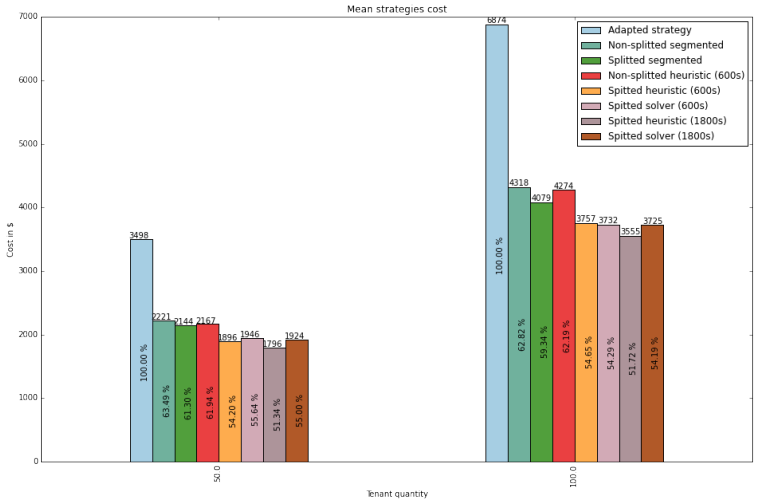

Fig. 6. Mean cost comparison for 50 and 100 tenants per group of 5

For 1800 seconds of execution time of the genetic algorithm, splitted heuristic give the best results. Mean distribution costs are 51.34 % for 50 tenants, and 51.72 % for 100 tenants of the naive coast. Using the solver gives good results but more expensive (respectively 55 % and 54.19 %). For 600 seconds of execution time, the results are more balanced : they vary between 54.2% and 55.64%. The genetic algorithm does not enhance the results a lot for both approaches after 600 seconds : 3% for the heuristic and less than 1 % for the solver. Still,

it enhances the initial splitted segmented results from 61.3 % to 51.34% for 50 tenants, and from 59.34 % to 51.72 % for 100 tenants.

We observe that the splitted segmented approach allows to gain more than 2 % , and to unleash the results of the genetic algorithm. Without splitting we gain of around 1 % for 600 seconds of genetic algorithm compared to the original population (non splitted segmented). When splitting, the genetic algorithm results in a gain of 7.1 % for 50 tenants, and 4.69 % for 100 tenants compared to the splitted segmented strategy. The absolute gain compared to the adapted heuristic remains worthwhile : we save 1702 $ for 50 tenants and 3319 $ for 100 tenants for a cost of respectively 3498$ and 6874$. The respective gain compared to our previous work is 425 $ and 763 $. For two days of operations, it could save 11445 $ per month for 100 tenants.

5 Conclusion

In this paper, we proposed a method for cost optimization of BPMaaS deployment based on tenant migration strategies and a genetic algorithm. We presented a new integer programming optimization model. Both allows to obtain substantial gains for BPMaaS providers. The result we obtain when we group the tenants is surprising. It may be explained by the size of the objective space. The fact that it is reproducible for different number of tenants shows that testing multiple sizes may allow providers to save on the operation.

Our method can probably be used with other metrics than BPM task throughput, if they can be expressed as a scalar for both the cloud resources and the tenants. We can consider for instance the number of processes, or the number of HTTP requests that lead to transactional processing. Our methods could be generalized on systems non related to BPMS using multi-tenancy and tenant-related persisted data.

Next steps include a better tuning of the genetic algorithm, that can be reached by auto-tuning methods like hyper-parameter optimization. We have not tested variations of the genetic algorithm parameters apart of some intuitive guesses and a selection of the best mutation rate, and it could certainly enhance the results. We also want to study more realistic customer profiles showing seasonality.

6 Acknowledgment

The authors would like to thank Gurobi for the usage of their optimizer, and Amazon Web Services for the EC2 instances credits (this paper is supported by an AWS in Education Research Grant Award).

References

1. Challenge ROADEF/EURO 2012 : Machine Reassignment, http://challenge.roadef.org/2012/en/sujet.php

2. Brandt, F., Speck, J., Völker, M.: Constraint-based large neighborhood search for machine reassignment: A solution approach to the ROADEF/EURO challenge 2012. Annals of Operations Research (Dec 2014)
3. Das, S., Nishimura, S., Agrawal, D., El Abbadi, A.: Live database migration for elasticity in a multitenant database for cloud platforms. CS, UCSB, Santa Barbara, CA, USA, Tech. Rep 9, 2010 (2010)
4. Djedović, A., Žunić, E., Avdagić, Z., Karabegović, A.: Optimization of business processes by automatic reallocation of resources using the genetic algorithm. In: Telecommunications (BIH), 2016 XI International Symposium on. pp. 1–7. IEEE (2016)
5. Euting, S., Janiesch, C., Fischer, R., Tai, S., Weber, I.: Scalable Business Process Execution in the Cloud. In: 2014 IEEE International Conference on Cloud Engineering (IC2E). pp. 175–184 (Mar 2014)
6. Garrett, A.: inspyred: Bio-inspired Algorithms in Python — inspyred 1.0 documentation (2014), http://pythonhosted.org/inspyred/
7. Gavranović, H., Buljubašić, M., Demirović, E.: Variable Neighborhood Search for Google Machine Reassignment problem. Electronic Notes in Discrete Mathematics 39, 209–216 (Dec 2012)
8. Gurobi Optimization, I.: Gurobi Optimizer Reference Manual (2015), http://www.gurobi.com
9. Hoenisch, P., Schuller, D., Schulte, S., Hochreiner, C., Dustdar, S.: Optimization of Complex Elastic Processes. IEEE Transactions on Services Computing 9(5), 700–713 (Sep 2016)
10. Janiesch, C., Weber, I., Kuhlenkamp, J., Menzel, M.: Optimizing the Performance of Automated Business Processes Executed on Virtualized Infrastructure. pp. 3818–3826. IEEE (Jan 2014)
11. Lovrić, M., Milanović, M., Stamenković, M.: Algoritmic methods for segmentation of time series: An overview. Journal of Contemporary Economic and Business Issues 1(1), 31–53 (2014)
12. Mitchell, S., OSullivan, M., Dunning, I.: PuLP: a linear programming toolkit for python. The University of Auckland, Auckland, New Zealand, http://www.optimization-online. org/DB_FILE/2011/09/3178. pdf (2011)
13. Rekik, M., Boukadi, K., Assy, N., Gaaloul, W., Ben-Abdallah, H.: A Linear Program for Optimal Configurable Business Processes Deployment into Cloud Federation. pp. 34–41. IEEE (Jun 2016)
14. Rosinosky, G., Youcef, S., Charoy, F.: An Efficient Approach for Multi-tenant Elastic Business Processes Management in Cloud Computing Environment. pp. 311–318. IEEE (Jun 2016)
15. Rosinosky, G., Youcef, S., Charoy, F.: Efficient migration-aware algorithms for elastic BPMaaS (2017)
16. Schulte, S., Janiesch, C., Venugopal, S., Weber, I., Hoenisch, P.: Elastic Business Process Management: State of the art and open challenges for BPM in the cloud. Future Generation Computer Systems (2014)
17. Whitley, D.: A genetic algorithm tutorial. Statistics and computing 4(2), 65–85 (1994)
18. Xu, J., Liu, C., Zhao, X., Yongchareon, S., Ding, Z.: Resource Management for Business Process Scheduling in the Presence of Availability Constraints. ACM Transactions on Management Information Systems 7(3), 1–26 (Oct 2016)